Funktionsobjekte

- Gliederung
- Einleitung STL/Algorithmen
- Funktionen als Parameter
- Ein bischen dynamischer: Templates
- Richtig dynamisch: Funktionsobjekte
- STL Parameterübergabe – Konsequenz für FO's
- Vordefinierte Funktionsobjekte
- Funktionadapter
- Zusammenfassung / Einschätzung

Einleitung

- STL bietet :
  - generische Container z.B. #include <vector>
  - allgemeine Algorithmen #include <algorithm>
    - std::sort, std::reverse, std::min_element, std::copy

- Freie Kombination von Container & Algorithmus
- Effizienz: Algorithms vs. Elementfunktionen

Funktionen als Parameter

- Einige Algorithmen erlauben Funktionen als Übergabeparameter z.B.

  for_each(container.begin() , container.end() , foo);
  - *foo(..)* mit jedem Element aufgerufen [addiert 10 und gibt aus]
  - Modifizieren, ausgeben usw.

- Funktionen i.A. stateless → verhalten sich immer gleich
- andere Zahlen als 10 addieren ?

Template Funktionen

- Wenn Wert zur Compilezeit bekannt:

```
template <int off>
void foo (int elem) {
    std:cout << elem+off << ' ';
}
```

- Gebrauch:

```
for_each(menge.begin() , menge.end() , foo<5>);
[for_each(menge.begin() , menge.end() , (int(*)(int)) foo<5>); ]
```

Template Funktionen (2)

- Parametrisierung muss zur Compilezeit feststehen
- Keine Konfiguration zur Laufzeit
- Aggregation von Informationen nicht möglich
  Kein Aufsummieren etc. (Stateless)

- Lösung wären Objekte, sie haben einen Zustand und können zur Laufzeit konfiguriert werden ...

Funktionsobjekte

- Objekte, die sich wie Funktionen “verhalten”

```
template <class Iterator, class Operation>
Operation for_each(Iterator start, Iterator ende , Operation op) {
    while (start != ende) {
        op(*start);
        start++;
    }
}
```

- Funktionsobjekte sind Objekte von Klassen mit selbst definiertem *operator ()*.

Funktionsobjekte (2)

- Bsp. 1

```
class FooClass {
        const int offset;
public:
        FooClass(int off) : offset(off) {};
        void operator() (int value)  {
                cout << value + offset << ' ';
        }
};
```

- Benutzung:

```
int some_offset = calculateOffset();
for_each(menge.begin() , menge.end() , FooClass(some_offset) );
```

Funktionsobjekte (3)

- Bsp. 2

```
class FooClass {
        int sum;
public:
        FooClass() : sum(0) {};
        int operator() (int value)  {
                sum+=value;
                return sum;
        }
};
```

- Benutzung:

```
FooClass foo;
tricky_algorithm(tree.begin() , tree.end(), result_vector , foo);
```

STL Parameterübergabe

- In STL durchgängig *call by value*
  oder *call by const. ref.*
- mit Konstanten nutzbar ; temporäre Objekte möglich
- → Funktionsobjekt wird kopiert
  → Zustandsänderung in der Kopie !

- Funktionsobjekte lagern Zustand aus, halten nur Zeiger/Referenz auf Zustand
  (elegant mit Ref-Counting verbinden)

Vordefinierte Funktionsobjekte

- *transform(menge_1.begin() , menge_1.end() , menge_2.begin() , menge_3.begin() , multiplies<int>());*
- Multipliziert jeweils zwei *ints* aus *menge_1* [ab *menge_1.begin()*] und *menge_2* [ab *menge_2.begin()*] und trägt Ergebnis ab *menge_3.begin()* ein, bis *menge_1.end()* errreicht ist. (parametrisiertes Funktionsobjekt)

- negate<typ>(), modulus<typ>(), equal_to<typ>() …
- plus<typ>(), minus<typ>(), multiplies<typ>() …
- less<typ>(), greater<typ>(), less_equal<typ>() …
- logical_not<typ>(), logical_and<typ>() …

Funktionsadapter

- *multiplies<int>()* wurde im Beispiel mit zwei Parametern "versorgt" – andere Algorithmen rufen Operation nur mit einem Parameter auf
- Funktionadapter binden einen Parameter eines Funktionsobjektes und erlauben damit die Verwendung in Situationen mit nur einem Parameter.
- Bsp.

  x = calculateX();

  pos = find_if(menge.begin() , menge.end(), ***bind2nd(****less<int>(), x****)*** *);*

Funktionsadapter (2)

- bind1st ,bind2nd
- not1, not2
- Werden eigene Funktionsobjekte von *unary_function* bzw. *binary_function* abgeleitet, kann man sie auch mit den Functionsadaptern verwenden.
- Signatur *bind2nd:*

  template <class Operation, class T>

  binder2nd<Operation> ***bind2nd****(const Operation& op, const T& x);*

- Returns:

  binder2nd<Operation>(op, typename Operation::second_argument_type(x)).

Funktionsadapter (3)

```
template <class Operation>
class binder2nd : public unary_function <typename
    Operation::first_argument_type, typename
    Operation::result_type> {
protected:
    Operation op;
    typename Operation::second_argument_type value;
public:
    binder2nd(const Operation& x,const typename
    Operation::second_argument_type& y);
    typename Operation::result_type operator()(const typename
                                Operation::first_argument_type& x) const;
};
```

Zusammenfassung

- Die STL bietet Aufruf von Algorithmen, parametrierbar mit Funktionen oder Funktionsobjekten.
- Funktionale Objekte sind mächtiger als „echte“ Funktionen.
- Funktionale Objekte haben selbst definierten *operator ()*.
- *call by value/const ref*. Semantik erfordert evtl. “ausgelagerten” Zustand.
- Typsicherheit zur Compilezeit
- Keine weiteren Anforderungen an Containerelemente

Wertung

- gut in generischem Umfeld (STL)
- Typsicher, flexibel zur Laufzeit

- Funktionsobjekte bekennen sich zu keinem Interface, keinem „Vertrag“.
- in Anwendungscode würde ich lieber Vererbung & virtuelle Funktionen verwenden.